



UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

INGENIERÍA INFORMÁTICA

Aprendizaje Automático para la extracción de características y detección de situaciones anómalas en multitudes

Autor

Diego Navarro Cabrera

Directores

Name of the main supervisor

Name of the second supervisor (if available)



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA Y
TELECOMUNICACIONES

—
Granada, 31 de mayo de 2021

Aprendizaje Automático para la extracción de características y detección de situaciones anómalas en multitudes

Diego Navarro Cabrera

Palabras clave:

Resumen

Machine learning for feature extraction and abnormal crowd behavior

Diego Navarro Cabrera

Keywords:

Abstract

Yo, **Diego Navarro Cabrera**, alumno del Grado de Ingeniería Informática de la **Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones de la Universidad de Granada**, con DNI 75935043Z, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

A handwritten signature in blue ink, appearing to read "Diego", enclosed within a roughly oval-shaped outline.

Fdo: Diego Navarro Cabrera

Granada, 31 de mayo de 2021

D. Name of the main supervisor y D.^a Name of the second supervisor (if available), profesores del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado *Aprendizaje Automático para la extracción de características y detección de situaciones anómalas en multitudes*, ha sido realizado bajo su supervisión por **Diego Navarro Cabrera**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 31 de mayo de 2021

Los directores:



Name of the main supervisor



**Name of the second supervisor
(if available)**

Agradecimientos

Índice general

Índice general	5
1 Introducción	7
2 Fundamentación Teórica	9
2.1. Aprendizaje automático	9
2.1.1. Tipos de tareas	9
2.1.2. Tipos de aprendizaje	10
2.1.3. Diseño de un modelo y generalización	11
2.2. Extracción de características	14
2.3. Máquina de soporte vectorial	15
2.4. Redes neuronales	17
2.5. Autoencoders	18
2.6. Aprendizaje multi-tarea	19
3 Descripción del problema	22
3.1. Conjuntos de datos de trabajo	22
3.1.1. UMN Crowd Dataset	23
3.1.2. Violent Flows	24
3.2. Estado del arte	25
3.2.1. Modelo basado en descriptores visuales locales de nivel medio	26
3.3. Modelo propuesto	28
3.3.1. Representación de la multitud	28
3.3.2. Descriptores visuales	31
3.3.3. Vector de características y clasificador	35
3.4. Aspectos de implementación	39
3.5. Consideraciones éticas	40
4 Resultados experimentales	42
4.1. Metodología usada	42

4.1.1. UMN Crowd Dataset	44
4.1.2. Violent Flows	46
4.2. Comparación de resultados	46
4.2.1. UMN Crowd Dataset	47
4.2.2. Violent Flows	50
5 Conclusiones y trabajo futuro	53
5.1. Conclusiones	53
5.2. Trabajo futuro	54
Bibliografía	55

1 Introducción

En los últimos años, se ha notado un interés creciente en la amenaza que suponen el crimen y el terrorismo. Una de las mayores herramientas para combatir dichas amenazas es la videovigilancia, pero la dada cantidad creciente de cámaras necesarias para llevar a cabo este control, cada vez es más difícil tratar la información que estas aportan. Por otro lado, los grandes avances en los campos del aprendizaje automático y de la visión por computador nos permiten tratar cada vez mejor la información recabada por dichos sistemas de videovigilancia. Es por esto que el análisis de distintas tareas relacionadas con estos sistemas se ha constituido como un campo de estudio dentro de estas áreas. Algunos ejemplos de tareas dentro de este campo pueden incluir la detección de objetos peligrosos o la detección de anomalías, de la que hablaremos en este trabajo.

La detección de anomalías trata de analizar el comportamiento de los individuos de una multitud, o de la multitud en sí misma, con la intención de detectar cuando dicho comportamiento se sale de lo habitual. De esta forma se pretende de detectar situaciones que puedan ser peligrosas, como una pelea. Hablaremos en más detalle de esto en el apartado correspondiente del capítulo 3.

A continuación explicaremos la estructura que seguiremos en este trabajo. En el capítulo 2 se hace un repaso teórico de diversos aspectos del aprendizaje automático y de la visión por computador. En el primer apartado explicaremos algunos conceptos generales sobre el aprendizaje automático, como los tipos de tareas que trata o aspectos que hemos de tener en cuenta al diseñar un modelo. En el segundo apartado hablaremos de la técnica de visión por computador conocida como “extracción de características”, que busca encontrar los puntos relevantes de una imagen para facilitar su procesamiento. En el apartado 3 explicaremos las máquinas de soporte vectorial, modelo que usaremos como clasificador de nuestro modelo. En los apartados 4 y 5 hablaremos de las redes neuronales, y en concreto del tipo de red que

usaremos en nuestro modelo, los *autoencoders*, que formarán parte del procesamiento que hagamos de los datos de entrada. Finalmente, en el apartado 6 hablaremos del aprendizaje multitarea, un tipo de entrenamiento útil cuando queremos que nuestro modelo tenga más de una finalidad, así como una buena capacidad de generalización.

En el capítulo 3 trataremos distintos aspectos relacionados con el problema que nos ocupa. Empezaremos explicando un poco más en detalle el problema con el que estamos trabajando, así como el enfoque que le queremos dar. En el primer apartado explicaremos los conjuntos de datos que usaremos para probar nuestro modelo. En el apartado 2 hablaremos del estado del arte, y concretamente explicaremos el modelo en el que nos hemos basado para construir nuestra propuesta. En el apartado 3 explicaremos en detalle nuestro modelo. En el apartado 4 daremos detalles relacionados con la implementación, como el lenguaje de programación y las librerías usadas. En el apartado 5 destacaremos algunas consideraciones éticas a tener en cuenta a la hora de desarrollar o usar un modelo como el nuestro.

En el capítulo 4 explicaremos los experimentos realizados, así como los resultados obtenidos. Primero hablaremos de los experimentos realizados en cada conjunto de datos. Después, en el apartado 2 compararemos los resultados obtenidos entre las distintas propuestas que podamos tener y el trabajo original en el que nos basamos.

En el capítulo 5 exponemos nuestras conclusiones, así como posibles vías de investigación futura para seguir tratando el problema y mejorar los resultados.

Fundamentación Teórica

En esta sección hablaremos de la fundamentación teórica del trabajo. En la sección 2.1 daremos una visión general del aprendizaje automático o *machine learning*. La sección 2.2 estará dedicada a una de las principales técnicas de procesamiento de imágenes, que será de gran importancia para construir nuestro modelo. En la sección 2.3 explicaremos el tipo de modelo que usaremos como clasificador en el modelo final que propongamos. En la sección 2.4 hablaremos de los conceptos más generales de las redes neuronales, para más adelante explicar en la sección 2.5 un tipo de red concreto que usaremos en nuestra propuesta. La sección 2.6 la dedicaremos para explicar algunos aspectos del aprendizaje multitarea, que deberemos tener en cuenta para entrenar la red antes mencionada.

2.1. Aprendizaje automático

El aprendizaje automático es un campo de la inteligencia artificial que busca detectar patrones entre un conjunto de datos de ejemplo para poder estimar una función con la que llevar a cabo una tarea. En el libro *Machine learning* [1], Thomas M. Mitchell define un programa de aprendizaje automático de la siguiente manera: “un programa informático se dice que aprende de la experiencia (E) con respecto a una clase de tarea (T), con una medida de efectividad (P), si su efectividad en la tarea T, medida por P, mejora con la experiencia E”. La naturaleza de este campo está íntimamente ligada a la de la estadística, de donde salen gran parte de las técnicas de *machine learning*.

2.1.1. Tipos de tareas

El tipo de tareas que puede realizar un algoritmo de aprendizaje automático es bastante amplio, sin embargo estas generalmente se pueden reducir en 4 grupos distintivos:

- **Clasificación:** se trata de todas aquellas tareas en las que existe un conjunto de categorías definidas desde el principio, entre las cuales queremos clasificar un conjunto de instancias. En este tipo de casos la salida (el valor que devuelve la función) puede ser directamente un valor que corresponda a la categoría más probable, o bien puede tratarse de un conjunto de valores que representen las probabilidades de que la entrada pertenezca a cada una de las categorías. Un ejemplo de este tipo de tareas puede ser la clasificación de razas de perro a partir de sus fotografías.
- **Regresión:** en este caso la tarea de la función es predecir un valor numérico a partir de la entrada. Un ejemplo de este caso podría ser el de un modelo que prediga la evolución de la población de una especie animal a lo largo del tiempo.
- **Agrupamiento:** también conocido como *clustering*, se trata de modelos que buscan agrupar en clases un conjunto de instancias en función de sus características comunes. Este caso es similar al de clasificación, con la excepción de que no siempre conocemos el número de grupos en los que se dividen los datos. Cuando describamos nuestro modelo propuesto más adelante veremos un ejemplo de este tipo de tarea, en donde tendremos que agrupar un conjunto de puntos en función de su localización en el espacio.
- **Detección de anomalías:** una de las tareas menos habituales pero no por ello de menor importancia. Su objetivo es estudiar los datos en busca de aquellas instancias que se salgan fuera de lo común. En un principio, el trabajo que desarrollaremos más adelante podría tratarse desde este punto de vista, sin embargo, debido a razones que discutiremos más adelante, desarrollaremos nuestra propuesta como un sistema de clasificación binaria.

2.1.2. Tipos de aprendizaje

A la hora de aprender de los datos nos encontramos con dos enfoques diferentes: el aprendizaje supervisado y el aprendizaje no supervisado. El primer caso se da cuando entrenamos usando no solo ejemplos de valores de entrada válidos para nuestra función, si no que también le indicamos la salida que deseamos para cada una de esas entradas. Este tipo de entrenamiento pretende encontrar la relación entre la entrada y la salida, de forma que ante nuevas entradas podamos predecir la salida que les corresponde.

El segundo caso se da cuando únicamente entrenamos usando los datos de entrada. Este tipo de aprendizaje se centra en extraer información de las entradas en sí mismas, o bien de la relación entre estas. Dos ejemplos con los que trabajaremos más adelante son los algoritmos de *clustering*, que buscan agrupar los datos sin saber previamente a qué grupo pertenecen estos o incluso cuantos grupos debería haber, y los algoritmos de reducción de dimensionalidad como el PCA (Principal Component Analysis) y los *autoencoders*, de los que hablaremos en más profundidad más adelante.

2.1.3. Diseño de un modelo y generalización

Para entrenar y usar un modelo de aprendizaje automático se necesitan los siguientes elementos:

- Un conjunto de **datos** a partir de los cuales aprenderemos a estimar nuestra función. Es quizás el elemento más importante ya que cualquier sesgo o error presente en este elemento se verá muy posiblemente trasladado al modelo final.
- El **diseño** de la función que represente nuestro modelo. A este aspecto es a lo que nos referimos cuando hablamos de distintos tipos de algoritmos de aprendizaje automático, como las redes neuronales o las máquinas de soporte vectorial, así como a los detalles de diseño dentro de estos subgrupos.

Es importante señalar que en 1997, el matemático David Wolpert demostró lo que se conoce como el *No Free Lunch Theorem* [2], que indica que, dado un conjunto de problemas lo suficientemente grande, la efectividad media de todos los algoritmos que probemos será la misma. Dicho de otra manera, no tiene sentido pensar en un algoritmo de aprendizaje automático que supere a los demás en todas las situaciones, si no que hemos de buscar el más apropiado para el caso concreto que nos ocupe.

- Una **función de pérdida**, que guíe el entrenamiento de nuestro modelo para que este se ajuste mejor a la tarea que intentamos resolver. Esta función depende del tipo de tarea con el que estemos trabajando, ya que modelos de clasificación habitualmente trabajarán con valores categóricos, mientras que funciones de regresión usarán valores numéricos.

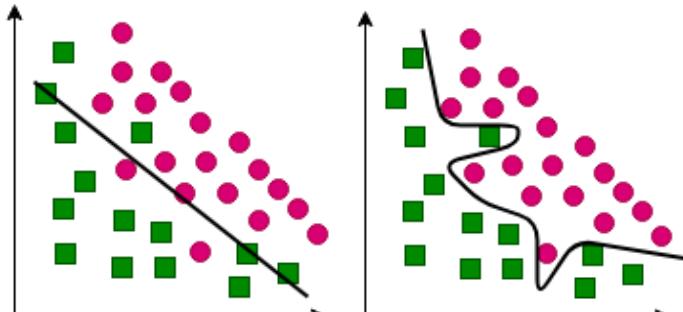
- Un **algoritmo de optimización**, que determine cómo ajustar los valores de nuestra función de tal manera que la función de pérdida alcance un valor óptimo.

Por otro lado, es muy importante tener en cuenta que nuestro objetivo es obtener un modelo que tenga una buena capacidad de generalización, es decir, que la función que obtengamos dé buenos resultados, no solo en los datos de entrenamiento, si no también cuando trabaja con datos nuevos, que el modelo no ha visto durante el entrenamiento. Este objetivo se puede dividir en 2, primero que nuestro modelo funcione correctamente en los datos de entrenamiento, y segundo, que se comporte de manera similar tanto dentro como fuera de dichos datos. Para esto generalmente dividimos los datos de los que disponemos en 2 grupos: datos de entrenamiento y datos de evaluación. De esta forma podemos entrenar el modelo y luego obtener una estimación lo menos sesgada posible de cómo debería comportarse nuestro modelo ante datos nuevos. Un problema de esta división es que cuantos más datos dediquemos a evaluar el modelo, menos tendremos para entrenar, y peor resultado obtendremos, pero si dedicamos pocos datos al testeo, la evaluación que obtengamos no será fiable. Para solucionar este problema se suele usar la validación cruzada, que consiste en repetir el proceso de división, entrenamiento y evaluación repetidas veces, para así poder usar conjuntos de entrenamiento más grandes, manteniendo la fiabilidad de la estimación, y además reducir el riesgo de sesgos dependientes de la división entre datos de entrenamiento o de *test*.

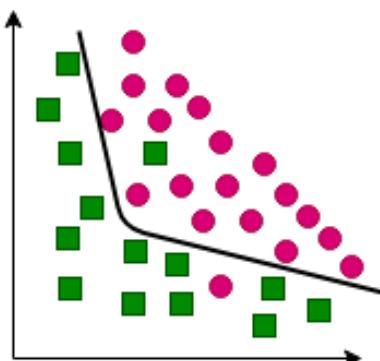
Dos conceptos importantes a la hora de buscar un modelo con buena generalización son el *overfitting* y el *underfitting*. El concepto de *underfitting* es bastante sencillo, se trata de obtener un modelo que no se ajuste adecuadamente a los datos, obteniendo una mala precisión incluso en el conjunto de entrenamiento. Esto puede deberse a muchas razones, como utilizar un modelo con poca capacidad expresiva (por ejemplo un modelo lineal) en un conjunto de datos complejo.

Contrario al *underfitting*, a veces nuestro modelo puede sufrir de *overfitting*. La raíz de este problema está en que siempre cabe esperar cierto nivel de *ruido* en nuestros datos, es decir, ligeras variaciones e inexactitudes que pueden provocar que no exista una división clara o una fórmula exacta para resolver la tarea que estemos tratando. Si nos centramos en optimizar la métrica del conjunto de entrenamiento, es posible que nuestro modelo aprenda del ruido presente en este, de forma que cuando se le presentan nuevos da-

tos nuestra clasificación estará sesgada por el ruido aprendido y tendrá una precisión menor. Este problema suele aparecer en casos donde tenemos pocos datos, ya que al usar un conjunto de entrenamiento pequeño es difícil extraer patrones generales que funcionen correctamente ante nuevos datos.



(a) Ejemplo de *underfitting*. (b) Ejemplo de *overfitting*.



(c) Ejemplo de clasificación con buena capacidad de generalización.

Figura 2.1: Ejemplos gráficos de *underfitting* y *overfitting* en un problema de clasificación binario.

En 2.1 podemos ver un ejemplo de estos dos conceptos que estamos tratando. Cuando usamos una función de clasificación (la línea separadora) demasiado simple, no somos capaces de dividir el conjunto de datos adecuadamente, sin embargo, si intentamos clasificar correctamente todos los elementos obtenemos una función muy compleja que tendrá problemas a la hora de clasificar nuevas instancias cercanas a la frontera de división.

Para aliviar el problema del *overfitting* se suelen usar medidas de regu-

larización, que buscan imponer restricciones en la función del modelo que a menudo se han demostrado útiles para la generalización. Más adelante, cuando hablemos del aprendizaje multitarea, desarrollaremos en más profundidad el tema del *overfitting* y la regularización.

2.2. Extracción de características

En este trabajo nuestro conjunto de datos estará formado por vídeos. Procesar toda la información de dichos vídeos puede ser muy costoso computacionalmente y a menudo llevarnos a error, dado que objetos distintos pueden compartir similaridades entre sí. Es por esto que a la hora de trabajar con vídeo a menudo se buscan primero zonas distintivas en la imagen que puedan darnos una mejor idea de lo que se está viendo en ella.

Es importante señalar que el término de “extracción de características” puede tener un significado ligeramente distinto en función de si lo usamos en el contexto de la visión por computador o en solo el del *machine learning*. En el segundo caso hablamos de las transformaciones que se llevan a cabo con datos de entrada antes de procesarlos en el modelo, pero en nuestro caso usaremos el significado que se le da en visión por computador y que ya hemos explicado anteriormente.

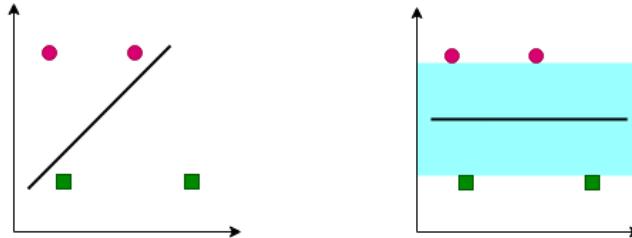
Existen múltiples formas de buscar características en una imagen, aunque en nuestro caso nos centraremos en aquellas que tienen como objetivo ser fáciles de rastrear. Para este tipo de características se suelen usar las esquinas de los objetos dentro de la imagen, ya que suelen coincidir con zonas en las que hay un claro contraste entre el objeto y lo que hay detrás, y si el objeto se mueve es fácil identificar la dirección y la distancia que ha recorrido.

Uno de los detectores más usados, y que usaremos en este trabajo debido a su alta velocidad de ejecución, es el algoritmo *Features from Accelerated Segment Test* (FAST) [3]. Este algoritmo tiene un funcionamiento muy sencillo que consiste en comparar un pixel central p con los 16 píxeles que lo rodean en un círculo de radio 3. El punto p es una esquina si la intensidad de un conjunto contiguo de esos píxeles es mayor/menor que la intensidad de p más/menos un umbral. Este algoritmo tan sencillo ha dado muy buenos resultados desde su creación, y dada su alta velocidad de ejecución se ha posicionado como uno de los más usados en su campo.

2.3. Máquina de soporte vectorial

El algoritmo de *machine learning* que usaremos en este trabajo para clasificar instancias se conoce como *máquina de soporte vectorial*, a partir de ahora SVM. Este es un algoritmo de clasificación que modifica la concepción de clasificador lineal, añadiendo 2 elementos que lo vuelven más robusto y lo dotan de una mayor capacidad para ajustarse a los datos.

Podemos definir un clasificador lineal como un hiperplano (una línea recta si pensamos en 2D) que divide el espacio de los datos en 2. Esto se puede ver en el ejemplo de underfitting que vimos en la figura 2.1. Este tipo de modelos tienen 2 problemas importantes: primero, en muchos casos una linea recta es incapaz de hacer una buena división de los datos, como se puede ver en el ejemplo mencionado; y segundo, a menudo existen infinidad de líneas que dan lugar a la misma división de los datos de entrenamiento, pero que tienen distinta posición e inclinación, y por lo tanto una mejor o peor capacidad de generalización. Podemos ver un ejemplo de la segunda problemática en 2.2



(a) Clasificador con una frontera (b) Clasificador lineal que maximiza la división demasiado cercana de miza el espacio entre los datos y la frontera de división.

Figura 2.2: Ejemplo de clasificadores lineales con el mismo valor de error en el conjunto de entrenamiento pero orientaciones distintas.

Para resolver el primer problema podemos transformar el espacio de los datos a uno de mayor dimensionalidad, en el que los datos sean linealmente separables, tal y como podemos ver en 2.3. Un problema de este método es que debemos calcular y operar con una mayor cantidad de datos, lo que aumenta el peso computacional de nuestro modelo, sin embargo las SVM hacen uso de lo que se conoce como el truco del *kernel*, que sin entrar en detalles matemáticos más allá del alcance de este trabajo, consiste en usar un tipo de funciones concretas, conocidas como *kernels* que nos permiten obtener

un resultado similar sin necesidad de calcular las coordenadas concretas del espacio de mayor dimensionalidad. Este tipo de funciones nos permiten trabajar en espacios de dimensiones mucho mayores que el original sin perder mucha eficiencia en los cálculos, lo que nos da lugar a modelos con una gran capacidad expresiva pero que también pueden dar lugar a *overfitting* si no se tiene cuidado.

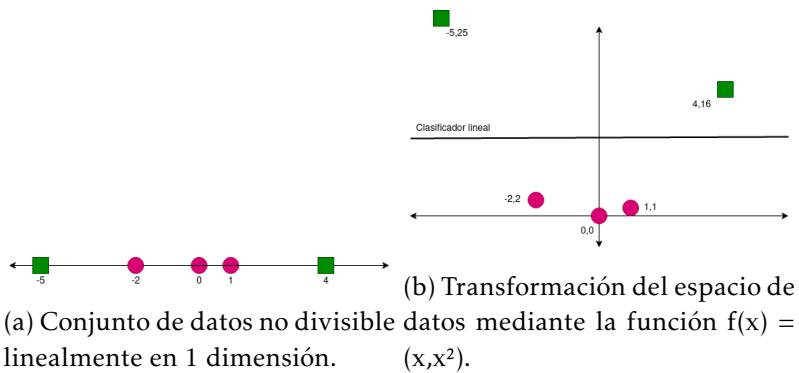


Figura 2.3: Ejemplo de cómo se puede modificar el espacio de los datos para poder dividir los datos con un clasificador lineal.

En cuanto al segundo problema de los clasificadores lineales, la solución es relativamente sencilla, ya que solo hay que buscar el clasificador que maximice la distancia hasta las instancias más cercanas. Existen dos enfoques para esto, las SVM de margen duro buscan maximizar la distancia hasta la instancia más cercana, mientras que las de margen blando permiten que algunas instancias de los datos se acerquen a la frontera de división, e incluso que estén mal clasificadas. Este último enfoque suele ser más adecuado, debido a la presencia de ruido en nuestros datos.

En general, cuando combinamos el truco del *kernel* para una mayor capacidad para separar los datos, con un margen blando para una mejor generalización, obtenemos un tipo de modelo muy potente, a la par que rápido. Este tipo de modelos es de los más usados en el ámbito del *machine learning*, solo superado por las redes neuronales profundas que han visto un enorme desarrollo en los últimos años.

2.4. Redes neuronales

Otra forma de superar las limitaciones de las funciones lineales es combinar varias de ellas en un solo modelo. Podemos ver un ejemplo de esto en 2.4, en donde usamos 2 funciones lineales para clasificar un conjunto que no podríamos dividir correctamente con una sola. De esta combinación de funciones es de donde sale el concepto detrás de las neuronas y redes neuronales.

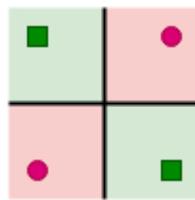


Figura 2.4: Ejemplo de un conjunto no separable linealmente y de cómo se podría dividir usando más de un hiperplano.

En *machine learning*, una neurona es la combinación de una función lineal (suma de las entradas ponderadas) y una función de activación no lineal, aplicadas a un conjunto de valores de entrada. Podemos ver un esquema de esto en 2.5. Una red neuronal será un conjunto de estas neuronas conectadas entre si.

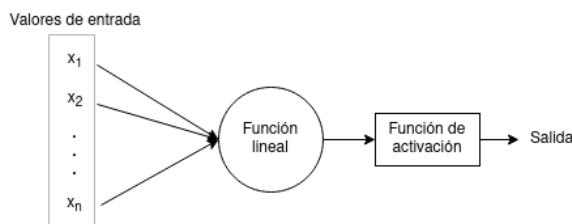


Figura 2.5: Esquema de una neurona.

La función de activación es un elemento muy importante, ya que sin ella la red sería equivalente a un modelo lineal normal y corriente. Existen múltiples funciones ampliamente usadas, pero la más extendida es la conocida como ReLU (*Rectified Linear Unit*) que se define como $\text{ReLU}(x) = \max(0, x)$. La elección de una función de activación es una cuestión de diseño difícil de resolver de manera teórica y a menudo se tiene que determinar de manera experimental cual es la que mejor se ajusta al problema.

Estas redes suelen ir estructuradas en capas, tal y como podemos ver en 2.6. La razón de esta estructura es que las neuronas de una misma capa pueden extraer distintos tipos de información de los datos que reciben. Conforme avancemos por la red habrá menos información, dado que habrá menos neuronas, pero esta estará más elaborada y será más útil, hasta que lleguemos a la última capa cuya salida es la solución de la tarea que estamos intentando resolver.

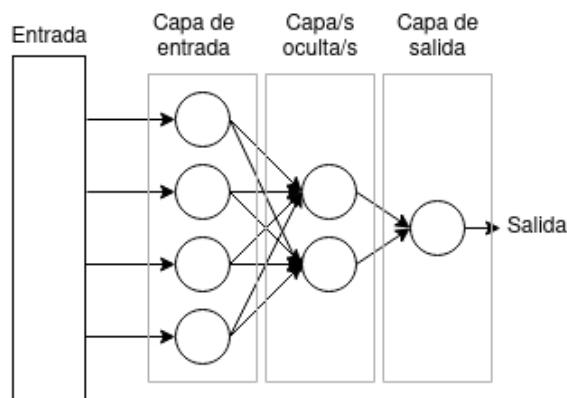


Figura 2.6: Estructura de una red neuronal.

Cuando contamos con multiples capas ocultas hablamos de aprendizaje profundo, o *deep learning*, que es un campo que ha despertado gran interés en los últimos años.

2.5. Autoencoders

Otro tipo de modelo de *machine learning* que usaremos en este trabajo son los *autoencoders*. Estos son un tipo de red neuronal que se usa a menudo para codificar los datos en un formato más reducido con el que sea más fácil trabajar. La estructura de esta red tiene 2 partes: un codificador, que transforma el vector de entrada en uno de un tamaño de un tamaño menor; y un decodificador, que usa la salida del codificador para intentar reconstruir la entrada original.

La idea detrás de este tipo de modelos es que los datos de entrada tienen una dimensionalidad mucho más alta de lo que se necesita para representar la misma información, por lo que es posible reducir dicha dimensionalidad de forma que cada elemento del vector aporte más información. Este nuevo vector, más elaborado pero de un menor tamaño no solo puede sustituir al

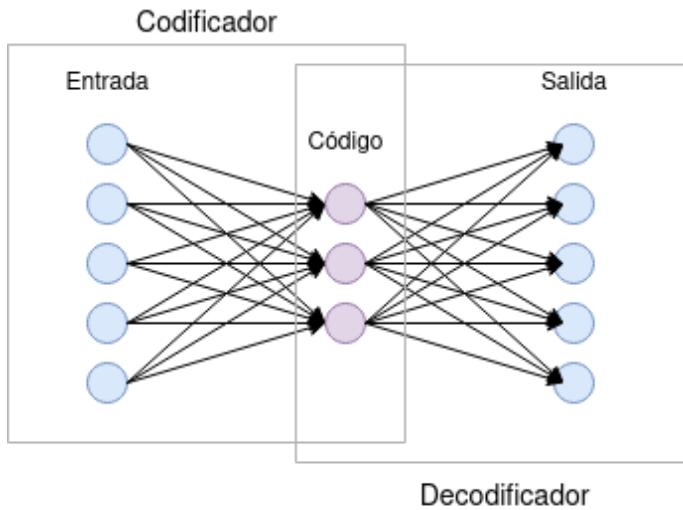


Figura 2.7: Diagrama de un autoencoder sencillo.

vector de características en un clasificador, lo que acelera el procesamiento de la información y ahorra espacio en memoria, si no que además a menudo ha demostrado ser más fácil de procesar, permitiendo hacer clasificaciones más fiables y precisas.

El entrenamiento de estos modelos se hace de manera no supervisada, ya que la salida esperada es la misma que la entrada. Esto hace que sean modelos fáciles de entrenar y aplicables a prácticamente cualquier problema, sin embargo, puesto que se siguen tratando de un tipo de red neuronal se suelen necesitar una gran cantidad de datos para entrenar correctamente estos modelos.

2.6. Aprendizaje multi-tarea

En ocasiones usamos modelos de *machine learning* con más de un objetivo, por ejemplo, más adelante usaremos un *autoencoder* para reducir la dimensionalidad de nuestros datos y poder clasificarlos más fácilmente. En ese caso la función principal del *autoencoder* será codificar el vector de entrada con la mínima pérdida de información posible, sin embargo ese no es su único objetivo. Un vector puede codificarse de muchas maneras, pero puesto que nuestra intención es usar la salida como entrada de un clasificador, preferiremos aquellas codificaciones en las que las distintas clases sean fáciles de distinguir. Es aquí donde entra el aprendizaje multitarea

En ocasiones este tipo de aprendizaje se puede aplicar por medio de funciones de pérdida cuidadosamente diseñadas, pero en nuestro caso aprovecharemos la modularidad del diseño por capas de las redes neuronales. Al igual que para usar un *autoencoder* primero entrenamos un modelo compuesto por codificador y decodificador, y después solo usamos el codificador, también es posible crear un modelo con más de una salida que compartan una serie de capas iniciales. En 2.8 podemos ver un ejemplo de como sería una red con 2 salidas. Si usásemos este esquema para el autoencoder que hemos comentado antes, las capas compartidas coincidirían con el codificador, una de las salidas con el decodificador, y la segunda salida podríamos dedicarla a un clasificador sencillo que nos dé una idea de la dificultad que supone clasificar el código creado por las capas iniciales.

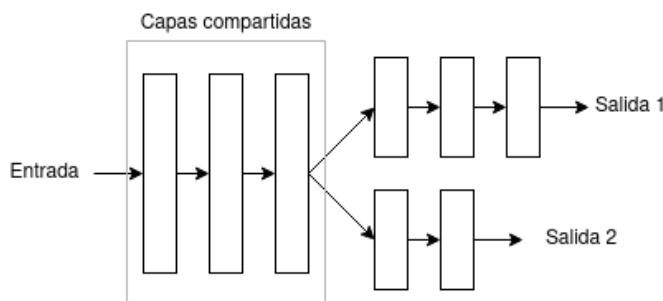


Figura 2.8: Esquema de una red neuronal multitarea.

Dada esta estructura modular, la función de pérdida que optimizamos durante el entrenamiento puede construirse sumando los valores de las funciones de pérdida de cada salida. Si quisieramos remarcar la importancia de una tarea por encima de otras podríamos ponderar el valor de cada función de pérdida, aunque esto no siempre es necesario. En nuestro caso mantendremos la misma ponderación para los dos sumandos de nuestra función de pérdida, el correspondiente a la clasificación y el correspondiente a la reconstrucción de los ejemplos.

Además de para adaptar el modelo para el problema que queremos que resuelva, el aprendizaje multitarea se ha demostrado muy útil para reducir el *overfitting*. Esto se debe a múltiples razones, pero una de las principales es que cuando una red tiene que buscar una representación que se ajuste bien para más de una tarea, el efecto que pueda tener el ruido sobre cada una de ellas es distinto, y por lo tanto será más fácil de filtrar.

Otra razón de por qué funciona el aprendizaje multitarea es que es más fácil identificar la información relevante, ya que esta será útil para más de una tarea, y puesto que cada tarea interactúa con los datos de una forma distinta, puede que haya información que sea más fácil de descubrir por medio de unas tareas o de otras.

3

Descripción del problema

En este capítulo desarrollaremos en más profundidad el problema a tratar. Como ya hemos mencionado anteriormente, nuestro objetivo es estudiar el comportamiento de una multitud para detectar cuándo se está produciendo una anomalía.

En concreto nos centraremos en el enfoque holístico o *top-down* de este problema. Este enfoque trata a la multitud como un solo ente y analiza la dinámica del grupo más que las acciones de sus individuos. La detección de comportamientos individuales anómalos dentro de la multitud queda fuera del ámbito de este trabajo, pero algunas propuestas de las estudiadas podrían adaptarse para ello.

La estructura de esta sección será la siguiente. Primero hablaremos de los conjuntos de datos sobre los que trabajaremos. A continuación trataremos el estado del arte y explicaremos en más profundidad el modelo sobre el que nos hemos basado. Finalmente detallaremos el modelo desarrollado y diversas alternativas que se han tenido en cuenta.

3.1. Conjuntos de datos de trabajo

Se han elegido 2 conjuntos de datos ampliamente usados en el ámbito de la detección de anomalías en multitudes. El primero es el UMN Crowd Dataset [4], que consiste de una pequeña selección de vídeos en el que un grupo de personas echa a correr tras una señal. El segundo conjunto de datos es conocido como Violent Flows [5] y recoge una mezcla de vídeos de peleas y vídeos de control. Estos dos conjuntos de datos nos permitirán comprobar la efectividad del modelo ante 2 de las anomalías más habituales y relevantes en el día a día: las peleas y los escenarios de pánico. Además su amplio uso en la literatura nos permitirá comparar fielmente nuestro modelo con otros muchos. A continuación describiremos más detalladamente ambos conjuntos.

3.1.1. UMN Crowd Dataset

Este conjunto de datos, elaborado por la Universidad de Minnesota (UMN), consiste de 11 vídeos en los que un grupo de personas se comporta de manera normal hasta que recibe una señal y todos se dispersan corriendo. Los 11 vídeos se dividen en 3 escenas distintas, cada una con 2, 6 y 3 vídeos respectivamente. Todos estos vídeos se encuentran unidos en un mismo archivo que conviene dividir antes de ser procesado.



Figura 3.1: Ejemplo de fotograma normal (izqda) y anómalo (drcha) en la escena 1.

Este dataset clasifica los fotogramas normales y los anómalos por medio de un cartel que aparece en pantalla cuando se produce la anomalía, tal y como se puede ver en 3.1. Para evitar que esto pueda influir en nuestro modelo se ha recortado la parte superior de cada vídeo para eliminar dicho cartel. Por otro lado, tal y como se detalla en [6] estos carteles no son exactos, ya que aparecen significativamente después del comienzo de la anomalía. Es por esto que se ha decidido usar las anotaciones provistas en [7], del que hablaremos más adelante, para etiquetar el comienzo de la anomalía en cada vídeo. Además de esto, en la mayoría de los casos la anomalía acaba antes de que empiece el siguiente vídeo, por lo que también tendremos que anotar el fotograma de finalización de cada caso. Hablaremos con más detalle de esto en el apartado de experimentación.

A pesar de sus ventajas y su uso muy extendido, este dataset presenta algunos inconvenientes que hay que tener en cuenta. Para empezar, su tamaño de 11 vídeos es bastante limitado, más aún si tenemos en cuenta que están divididos en 3 escenas que han de evaluarse de manera separada. Además solo representa anomalías de un tipo muy específico (gente echando a correr), por lo que no nos sirve para evaluar modelos enfocados a detectar todo tipo de

anomalías, algo de lo que hablaremos cuando propongamos nuestro modelo.

3.1.2. Violent Flows

Este conjunto de datos es una colección de vídeos de escenas reales para entrenar y evaluar modelos de detección de violencia. Contiene 246 instancias que se dividen en vídeos con violencia y sin ella. Se tratan de escenas cortas de entre 1 y 6 segundos y que se clasifican de manera íntegra como violentas o no. En la figura 3.2 podemos ver un ejemplo de un vídeo normal y uno anómalo.



Figura 3.2: Fotograma de ejemplo de una escena normal (izqda) y una violenta (drcha)

Los vídeos de este conjunto han sido extraídos de YouTube, y son muy distintos entre si, lo que nos permitirá comprobar la capacidad de generalización de nuestro modelo. Por otro lado, la baja resolución a la que se encuentran dificulta su análisis, ya que algunos fotogramas están tan borrosos que es difícil saber qué está pasando.

De forma similar al conjunto de datos anterior, este posee una cantidad de instancias de entrenamiento bastante limitada (246 vídeos clasificados individualmente), y se centra en un tipo de anomalías concreto, en este caso las peleas. Por otro lado, al contrario de lo que cabría esperar al trabajar con anomalías, ambas clases tienen un número de instancias muy equilibrado, 121 vídeos violentos y 125 normales, lo que nos podría facilitar el entrenamiento de un modelo de clasificación binaria, pero dificultar el uso de otras opciones centradas en entrenar únicamente con instancias normales (clasificadores de una sola clase).

3.2. Estado del arte

Existen numerosos trabajos que han probado diversas formas de detectar comportamientos anómalos. Algunas de estas son:

- Estudiar el flujo óptico de la imagen. Por ejemplo, en [8] usan el valor medio de este para estimar una interacción de fuerzas y detectar posibles zonas de conflicto, tal y como se ve en la figura 3.3.

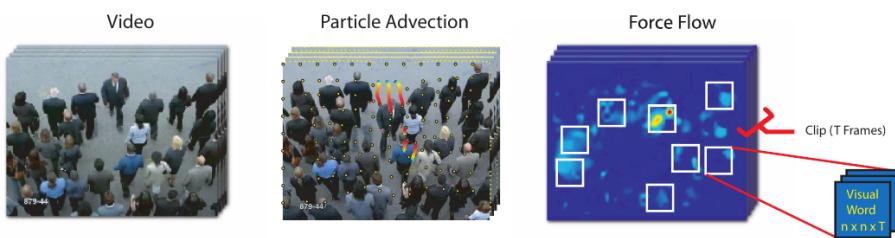


Figura 3.3: Demostración de la interacción de fuerzas, primero creamos un grid de partículas y estudiamos su trayectoria interpolando los movimientos cercanos a estas. Las zonas con mayor movimiento en una misma dirección se marcan como más fuertes, y en función de dichas fuerzas se detectan comportamientos anómalos.

- Codificar el vídeo por medio de un modelo entrenado con vídeos no anómalos. Puesto que dicho modelo solo conoce los vídeos normales, al intentar reconstruir una anomalía se obtendrá un resultado más irregular o con una mayor tasa de error, lo que nos marcará que se está produciendo una anomalía. Un ejemplo de esto se puede ver en [9] que usa un diccionario de códigos binarios y un histograma de dichos códigos como representación.
- Usar modelos *end-to-end* de *deep learning*, como en [10], donde se entrenan 2 redes adversarias: un generador y un discriminador, y usa el discriminador para clasificar cada fotograma como normal o anómalo.

En nuestro caso queremos centrarnos en un modelo que extraiga un conjunto de características de forma analítica, aprovechando el conocimiento y la intuición que ya tenemos sobre el tipo de anomalías esperadas. Dentro de los trabajos que siguen este enfoque el que parece aportar más información y mejores resultados es el que trataremos a continuación.

3.2.1. Modelo basado en descriptores visuales locales de nivel medio

El modelo sobre el que vamos a basar nuestra propuesta es el expuesto en [7]. Este artículo propone un proceso analítico de extracción de características, que se realiza a partir de una representación espacio-temporal de la multitud. En este apartado no daremos muchos detalles ya que estos los desarrollaremos en la descripción de nuestro modelo. En dicho apartado comentaremos no solo el funcionamiento de nuestro modelo, si no también los cambios que se han realizado con respecto a este y sus razones.

La representación espacio-temporal está formada por un conjunto de puntos de interés extraídos por un algoritmo de detección como FAST [3]. Los puntos son rastreados a lo largo del tiempo, guardando las trayectorias que siguen, representando así la información temporal del modelo. Para hacer esto se usa un rastreador de características dispersas como RLOF [11] o el rastreador de Kanade–Lucas–Tomasi [12].

Para representar la información espacial, cada punto está relacionado con sus vecinos por medio de una triangulación de Delaunay. Dicha triangulación es una red de triángulos conexa y convexa en donde la circunferencia circunscrita de cada triángulo no contiene ningún vértice de otro triángulo. También podría usarse un algoritmo de los K vecinos más cercanos, pero el artículo sostiene que el grafo aporta una mejor representación de la información local y espacial.

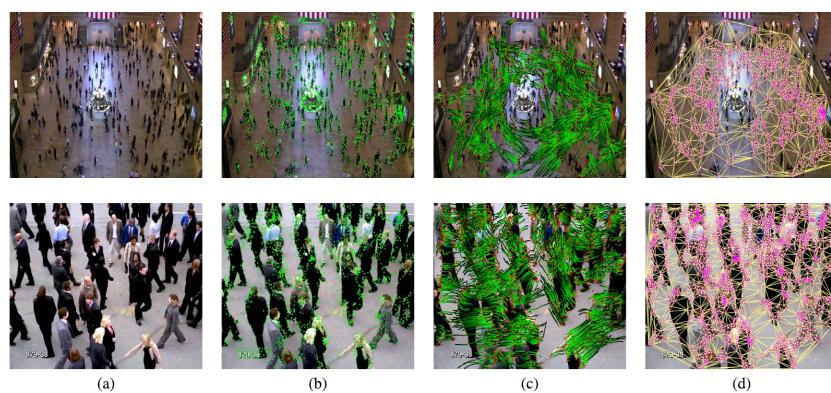


Figura 3.4: Ejemplo extraído de [7]. (a) Fotograma de ejemplo. (b) Detección de puntos de interés con FAST. (c) Trayectorias de los puntos a lo largo del tiempo. (d) Triangulación de Delaunay.

Una vez representada la información de la multitud, se calcula una serie de descriptores por cada punto detectado. Los descriptores son los siguientes:

- **Velocidad** de cada punto rastreado.
- **Variación del flujo dirección:** cuanto más recta sea la trayectoria, menor será este valor.
- **Estabilidad** de la estructura de la multitud: una multitud que mantiene una misma forma durante mucho tiempo será estable, pero una que se mueve de forma errática no.
- **Cohesión** (*Collectiveness*): grado en el que los individuos se mueven de manera conjunta y en una misma dirección.
- **Conflictos:** medición del nivel de interacción entre personas cercanas.
- **Densidad local:** número aproximado de personas en una determinada zona, calculado en función del número de puntos localizados en esa zona.
- **Uniformidad:** indicador de si los puntos rastreados se distribuyen de manera uniforme o se concentran en ciertas zonas formando subgrupos.

Una vez hechos todos los cálculos se generan histogramas para cada descriptor, se juntan para formar un vector de características y se entrena un SVM de clasificación binaria.

En el propio artículo podemos ver los resultados de este modelo en los 2 conjuntos de datos que vamos a usar, obteniendo una puntuación de 88%, usando la métrica del área por debajo de la curva (AUC), en el conjunto de Violent Flows, y un 98.6% en el de UMN. Estos resultados son especialmente buenos, sobre todo si nos fijamos en las comparaciones que se exponen en el mismo artículo con respecto a trabajos similares.

Por otro lado, también menciona la capacidad de este sistema de funcionar a tiempo real, aunque solo indica que siendo implementado en C++ funciona a 5 fotogramas por segundo “*sin estar optimizado*”. Esta afirmación es demasiado vaga como para ser tomada en cuenta, ya que la velocidad de este algoritmo depende en gran medida del número de puntos captados, algo que puede variar en gran medida en función del vídeo o de los parámetros elegidos. Más adelante expandiremos más sobre este tema.

3.3. Modelo propuesto

En esta sección hablaremos del proceso de implementar un sistema lo más cercano posible al descrito en el apartado anterior, así como los cambios que se han propuesto para mejorarlo. La mayoría de lo que se mencione en este apartado será común tanto al modelo descrito en [7], como a nuestro modelo, ya que estamos partiendo de este, pero se señalarán los casos en los que se diverja del diseño original.

El mayor obstáculo que se ha encontrado para intentar replicar este modelo es que aunque está lo suficientemente bien explicado como para entender su funcionamiento, a la hora de entrar en detalles de implementación, e incluso de evaluación sobre los conjuntos de datos, hay apartados en los que no disponemos de suficiente información como para replicar fielmente lo expuesto en el artículo. iremos mencionando estos casos conforme vayamos avanzando.

El modelo que proponemos se puede dividir en 3 partes: la representación inicial de la multitud, el cálculo de descriptores y el procesamiento de dichos descriptores. En 3.5 podemos ver en más detalle el proceso que sigue dicho modelo. En los siguientes apartados explicaremos cada parte con más detalle.

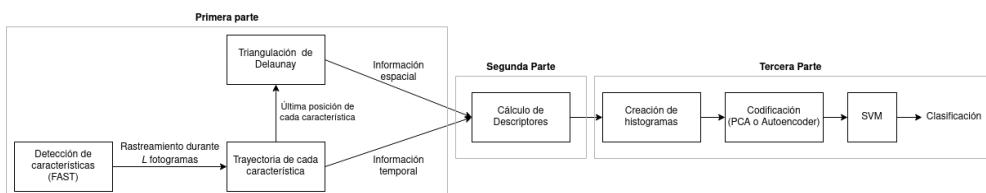


Figura 3.5: Esquema general del modelo propuesto.

3.3.1. Representación de la multitud

Como comentábamos en la sección anterior, el primer paso para detectar una anomalía es representar la información espacio-temporal de la multitud. Para esto primero detectamos un conjunto de puntos de interés mediante el algoritmo FAST (Features from Accelerated Segment Test) [3]. El fin de este algoritmo es buscar esquinas en la imagen, y dada la forma y apariencia de una persona, se detectará un gran número de estos puntos en cada persona. Para distinguir a los puntos que corresponden con una persona de los

del fondo de la imagen usaremos un filtro basado en su velocidad de movimiento. Puesto que la velocidad de cada punto es uno de los descriptores que vamos a usar, podemos usar este valor para eliminar todos los puntos que se encuentren por debajo de un umbral. Este filtro suele funcionar bien en escenas en las que la cámara está fija, pero como veremos más adelante este no siempre es el caso, e incluso hay factores que pueden complicar su buen funcionamiento aunque se cumpla esta condición.

Una vez tenemos un conjunto de puntos localizados usaremos el rastreador de Kanade–Lucas–Tomasi, que nos puede indicar la posición siguiente de un punto a partir de dos fotogramas consecutivos y su localización original. Para una mayor fiabilidad en el rastreo usaremos un esquema de verificación hacia delante y hacia atrás, es decir, después de obtener la posición final de un punto la verificaremos realizando el rastreo a la inversa, si obtenemos de nuevo la posición original (o una lo suficientemente cercana) tomaremos el resultado como bueno, y si no consideramos que hemos perdido la trayectoria y eliminaremos el punto.

La forma de gestionar las trayectorias será el primer aspecto en el que nos separaremos de [7]. En este artículo se menciona una alternancia entre la detección de nuevos puntos y el rastreo de los ya existentes en función de la fiabilidad de la trayectoria, sin embargo demasiados aspectos no se detallan, como la manera de incorporar estos nuevos puntos en el sistema sin sobrecargarlo de posiciones muy cercanas entre si, o el criterio para detectar a personas que hayan podido entrar en escena.

Es por eso que en su lugar se ha decidido seguir un esquema más sencillo en el que si usamos trayectorias de longitud L , primero llenamos la trayectoria del punto durante esos L fotogramas y luego calculamos los descriptores de ese punto durante otros L fotogramas. De esta forma, detectariamos nuevos puntos, como mucho, cada $2L$ fotogramas. Puesto que este método nos obligaría a renunciar a la información de la mitad de los fotogramas usaremos 2 conjuntos de trayectorias, una que usaremos para calcular los descriptores y otra en la que iremos construyendo la trayectoria de los nuevos puntos.

Cabe destacar que este segundo conjunto de trayectorias podría desecharse en una implementación orientada a un uso práctico en la que no fuese importante obtener toda la información de los vídeos, pero los conjuntos de datos con los que trabajamos son bastante pequeños y las precisiones lo bas-

tante altas como para que cada fotograma cuente.

En la figura 3.6 podemos ver 2 ejemplos de las trayectorias detectadas por nuestra implementación. En la primera se ve una situación normal, en donde la multitud se desplaza tranquilamente, mientras que en la segunda se ve como la gente echa a correr, como si huyesen de algo. Se puede apreciar que aunque el seguimiento de algunos puntos no es perfecto, la trayectoria general de cada persona está clara.



Figura 3.6: Trayectorias detectadas por el modelo en una situación normal (derecha) y una de pánico (izquierda).

Por otro lado, como decíamos en el apartado anterior, usaremos una triangulación de Delaunay para relacionar espacialmente los puntos de cada fotograma. OpenCV ya nos proporciona una clase que llevará a cabo este proceso, llamada SubDiv2D. Una vez creado el grafo en los siguientes apartados haremos uso del concepto de clique de primer orden para relacionar unos puntos con otros. Definimos el clique del punto V_i como todos los puntos V_j del grafo tales que existe una arista que une V_i y V_j . En [7] se habla también de los cliques de orden mayor que uno. Estos cliques se definen de manera recursiva de forma que un clique de orden k se forma añadiendo al clique de orden $k-1$ los puntos conectados con cualquier miembro este. Puesto que en ningún momento se menciona que estos aporten nada de valor al cálculo, y además añadirían un gran peso computacional, ignoraremos estos cliques de orden superior y usaremos solo cliques de primer orden.

En la figura 3.7 podemos ver el grafo que se obtiene al llevar a cabo la triangulación de Delaunay sobre los puntos detectados en uno de los fotogramas del conjunto de datos UMN. También se ha marcado el clique correspondiente a uno de los puntos, destacando dicho punto en rojo, y los adyacentes en azul.

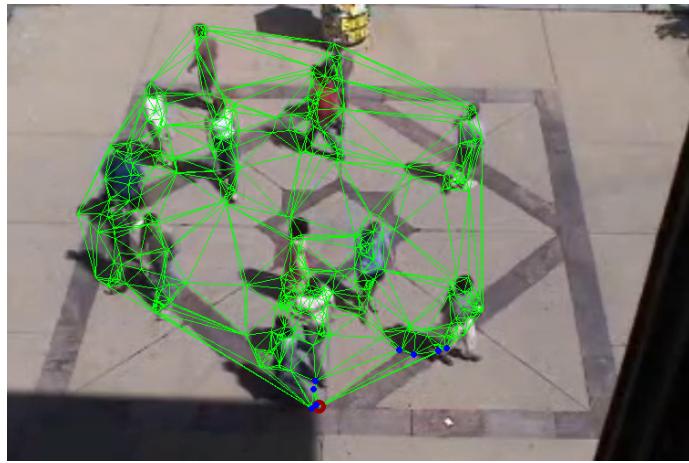


Figura 3.7: Triangulación de Delaunay en un fotograma de ejemplo (escena 3 UMN).

Resumiendo todo el proceso de este apartado nos quedamos con 3 pasos principales. Primero captar puntos de interés con FAST, que más adelante serán filtrados en función de su velocidad. Segundo, representar la evolución temporal de la multitud por medio de las trayectorias de cada punto a lo largo de un determinado número de fotogramas. Tercero, representar las relaciones espaciales de los puntos en cada fotograma por medio de los cliques de primer orden presentes en una triangulación de Delaunay que contenga a todos los puntos.

3.3.2. Descriptores visuales

En este apartado explicaremos en mayor profundidad los descriptores que usaremos para representar el estado de la multitud. Estos se calculan usando a partir de las representaciones explicadas en el apartado anterior. Existen 3 parámetros que determinarán que momento de la trayectoria usamos para realizar ciertos cálculos. El primero es el tamaño total de la trayectoria, que denotaremos como L . Puesto que estas trayectorias miden un periodo de tiempo relativamente amplio también usaremos otros dos parámetros, τ_1 y τ_2 , que marcarán momentos intermedios de la trayectoria que usaremos como referencia a la hora de calcular algunos valores como la velocidad.

- **Velocidad:** en [7] se calcula este valor como la distancia euclídea de la posición actual de un punto y su posición τ_1 fotogramas antes. Puesto que de la cámara no suele ser perpendicular al plano en el que se mueve

la multitud, los desplazamientos en el eje x e y tienen ordenes de magnitud distintos. Para solucionar esto [7] crea un mapa de perspectiva interpolando la altura de una persona de referencia en dos extremos de la imagen. Esta idea está insuficientemente detallada como para poder ser replicada con fidelidad, y además solo es aplicable a conjuntos de datos en los que los videos comparten una misma escena. Es por esto que en su lugar se ha decidido dividir este descriptor en 2 que se traten por separado, uno que mida la velocidad en el eje x y otro que la mida en el eje y . En cuanto al filtro que mencionábamos en el apartado anterior, una vez calculada la distancia que recorre un punto en el eje x e y , si la suma de ambos valores (distancia Manhattan) es menor que un valor concreto β , dicho punto es eliminado.

- **Variación del flujo de dirección:** este descriptor busca distinguir entre movimientos suaves y caóticos. Su cálculo será similar al de [7], con la única diferencia que nosotros solo usamos trayectorias de tamaño constante L , lo que nos permite ahorrarnos ciertos cálculos superfluos. Para este cálculo primero seleccionamos F fotogramas separados a una distancia de τ_2 entre sí. Dado que las trayectorias tienen un tamaño fijo de L fotogramas, $F = \lfloor L/\tau_2 \rfloor$. Una vez dividida la trayectoria, calculamos la dirección de cada fotograma seleccionado al siguiente y calculamos la diferencia con la dirección anterior. Este cálculo se puede expresar de la siguiente forma, siendo V_i^k el punto i en el instante de tiempo k , d_θ la diferencia angular (que definiremos a continuación), y $S_i^{k-f\tau_2}$ la posición que tenía el punto i en el instante $k - f\tau_2$:

$$D^{var}(V_i^k) = \sum_{f=0}^{F-2} d_\theta(S_i^{k-f\tau_2}, S_i^{k-(f+1)\tau_2}) \quad (3.1)$$

Definimos d_θ como $\Delta\theta(|\theta(a) - \theta(b)|)$, donde $\theta(a)$ es el ángulo que el vector a hace con el eje x . $\Delta\theta(\alpha)$ sirve para corregir la dirección de α y se define como α cuando este es menor o igual que π y $2\pi - \alpha$ cuando es mayor.

- **Estabilidad:** este descriptor intenta estudiar la estructura topológica de la triangulación de Delaunay a lo largo del tiempo. Para esto comparamos la forma de este en el instante actual, con la que tenía hace τ_2 fotogramas. En [7] se intenta tener en cuenta tanto la distribución de los cliques como los puntos que son eliminados e insertados en el gráfico, sin embargo esto supone una gran cantidad de elementos a guardar en

memoria (p.e. el grafo de cada fotograma) y de cálculos (volver a calcular los cliques de fotogramas pasados). En nuestro caso nos quedaremos solo con el elemento principal de este descriptor, la deformación de los triángulos, y asumiremos que los cliques están estructurados de la misma forma en ambos fotogramas. De esta forma, por cada clique de grado 1 podremos sacar un conjunto de triángulos $r_{i\alpha}$. Por cada triángulo que tengamos en un clique calcularemos su diferencia con el triángulo formado por la posición de esos tres mismos puntos τ_2 fotogramas antes. La diferencia de ambos triangulos se define de la siguiente manera:

$$g(r_{i\alpha}, r_{i\beta}) = |a_{i\alpha} - a_{i\beta}| * |c_{i\alpha} - c_{i\beta}| \quad (3.2)$$

Donde $a_{i\alpha}$ es el área del triangulo y $c_{i\alpha}$ es el *cross-ratio*, que se calcula con los dos vértices del triángulo que no son i ($V_\alpha, V_{\alpha'}$), y las proyecciones de los puntos medios de los lados que van a i ($p'_{i\alpha}, p'_{i\alpha'}$). Estos 4 puntos se ordenan en una misma dirección, formando un vector (x_1, x_2, x_3, x_4) con el que se calcula la *cross-ratio* de la siguiente manera:

$$f_{cr}(x_1, x_2, x_3, x_4) = \frac{|x_1 - x_3| * |x_2 - x_4|}{|x_1 - x_4| * |x_2 - x_3|} \quad (3.3)$$

Sumando el resultado de todos los triangulos de un clique obtenemos su estabilidad.

- **Cohesión:** este valor busca estudiar el grado en el que los individuos que forman la multitud se mueven en una misma dirección. Se calcula usando la siguiente fórmula:

$$D^{coh}(V_i^k) = \frac{1}{|C_n V_i^k|} \sum_{V_j^k \in C_n(V_i^k)} d_\theta(\overrightarrow{V_i^{k-\tau_1} V_i^k}, \overrightarrow{V_j^{k-\tau_1} V_j^k}) \quad (3.4)$$

Donde $C_n V_i^k$ es el conjunto de puntos pertenecientes al clique de orden 1 de V_i en el instante k . Esta fórmula es igual que la que se usa en [7] salvo por el uso de la función d_θ , que en su caso se compara primero con un valor T_1 y si es menor que este se sustituye por un 0. Puesto que en ningún momento se menciona qué es T_1 , por qué debería usarse o qué valor debería tomar se ha decidido prescindir de él. Contrariamente a lo que podríamos pensar, esta fórmula tomará valores más bajos conforme más cohesionada esté la multitud, y tomará valores más altos cuando las trayectorias de los puntos de un mismo clique sean muy distintas.

- **Conflictivo:** el objetivo de este descriptor es caracterizar la interacción entre los miembros de la multitud. Se calcula de manera muy similar a la cohesión, y de hecho podemos implementar ambos valores dentro de una misma función para evitar repetir cálculos innecesarios. Su fórmula es la siguiente:

$$D^{conf}(V_i^k) = \frac{1}{|C_n V_i^k|} \sum_{V_j^k \in C_n(V_i^k)} \frac{d_\theta(\overrightarrow{V_i^{k-\tau_1} V_i^k}, \overrightarrow{V_j^{k-\tau_1} V_j^k})}{\|V_i^k V_j^k\|_2} \quad (3.5)$$

Como se puede ver, la única diferencia es que en este caso dividimos por la distancia de V_i^k a V_j^k , lo que, como es de esperar, nos dará valores de conflicto más altos cuando los puntos estén cercanos entre si.

- **Densidad:** este valor describe la cercanía de los puntos de un mismo clique. se calcula de la siguiente forma:

$$D^{dens}(V_i^k) = \frac{1}{\sqrt{2\pi}\sigma} \sum_{V_j^k \in C_n(V_i^k)} \exp - \frac{\|V_i^k V_j^k\|_2}{2\sigma^2} \quad (3.6)$$

Donde σ es el ancho de banda del kernel Gaussiano que determina el efecto de cada vecino en la densidad. En ciertos casos este valor debería ajustarse mediante un mapa de perspectiva, pero puesto que en la mayoría de las escenas en las que trabajaremos no hay una gran distorsión de la perspectiva usaremos un valor constante que ajustaremos empíricamente.

- **Uniformidad:** este descriptor refleja la homogeneidad de la distribución de los distintos subgrupos dentro de una misma multitud. El primer paso para calcular este valor es usar un algoritmo de *clustering* para agrupar los puntos del grafo en grupos. En [7] no se menciona ningún algoritmo concreto, y solo se recomienda que se use un algoritmo basado en la distancia que no necesite conocer el número de grupos de antemano. Dentro de este tipo de algoritmos uno de los más usados y que mejor funciona es DBSCAN [13]. Una vez tenemos un conjunto de p grupos, $N = N_1, N_2, \dots, N_p$ calculamos la uniformidad de un grupo como:

$$D^{unif}(N_i) = \frac{A(N_i, N_i)}{A(N, N)} - \left(\frac{A(N_i, N)}{A(N, N)} \right)^2 \quad (3.7)$$

$$A(N_i, N_i) = \sum_{p \in N_i} \sum_{q \in C_1(p), q \in N_i} \frac{1}{\|pq\|_2} \quad (3.8)$$

$$A(N_i, N) = \sum_{p \in N_i} \sum_{q \in C_1(p), q \notin N_i} \frac{1}{\|pq\|_2} \quad (3.9)$$

$$A(N, N) = \sum_{i \in N} \sum_{p \in N_i} \sum_{q \in C_1(p)} \frac{1}{\|pq\|_2} \quad (3.10)$$

En resumen, la uniformidad de un grupo es alta si la distancia de los puntos de una misma clase es pequeña y la distancia entre puntos de distintas clases es grande. Por último, aunque este descriptor se calcula por grupos, para mantener la homogeneidad con el resto de descriptores le asignaremos a cada punto el valor de su grupo correspondiente. De esta forma también podremos tener en cuenta, de manera indirecta, el número de puntos de cada grupo cuando juntamos estos valores en un histograma.

En la figura 3.8 podemos ver un fotograma de ejemplo en el que se han marcado los puntos detectados, asignandole un color a cada grupo detectado por el algoritmo de *clustering*. Se considera que esta división se ajusta gratamente a lo que cabría esperar del problema planteado, y que por lo tanto el algoritmo de *clustering* funciona adecuadamente.

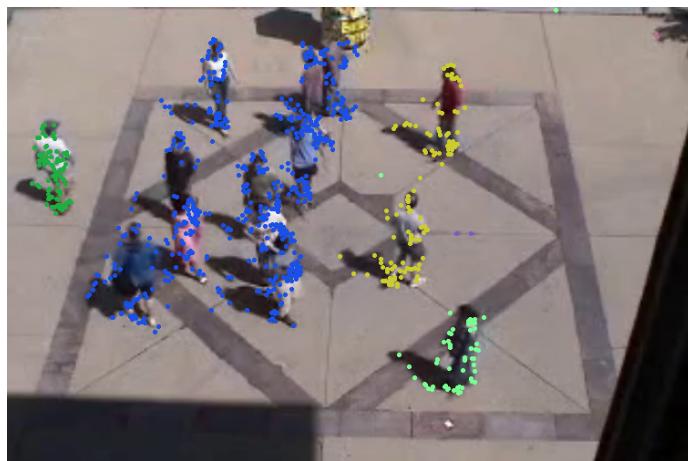


Figura 3.8: *Clusters* encontrados con DBSCAN en la escena 3 del conjunto UMN.

3.3.3. Vector de características y clasificador

Una vez hemos calculado los descriptores para cada punto rastreado solo nos queda procesar esta información mediante técnicas de aprendizaje automático para obtener una predicción. Puesto que el número de puntos puede

variar en cada fotograma, construiremos un histograma para cada descriptor en cada fotograma. Un histograma es una representación de la distribución numérica de un conjunto de valores dentro de un rango. Se construye dividiendo en n partes iguales el rango indicado, y contando el número de valores que entra en cada partición. Con esta representación podemos estimar si la distribución de valores es normal o si hay una cantidad anormal de valores altos o bajos. Para el rango de los histogramas usaremos 0 como valor mínimo, ya que es el valor mínimo de todos los descriptores, y como valor máximo usaremos el percentil 95 de los datos que usemos para entrenar. Usamos un percentil en lugar del máximo absoluto porque valores anórmalmente altos podrían afectar demasiado al rango provocando que la mayoría de los valores se concentren en el primer intervalo, perdiendo por tanto mucha precisión. Para evitar sesgos y facilitar el entrenamiento normalizaremos los valores de cada histograma en el rango [0,1]. En [7] se menciona usar $n = 16$, pero en nuestro caso se ha encontrado que 32 suele arrojar mejores resultados. Puesto que tenemos 8 descriptores (recordemos que la velocidad se divide en 2) usaríamos vectores de unos 256 elementos.

En [7] este vector se usa directamente con un SVM binario para entrenar el modelo o para predecir la clase de un fotograma o un vídeo. Hay que señalar que en el estudio de anomalías se suelen usar también otro tipo de clasificadores, como los de una sola clase, pero debido a algunos de los problemas que comentábamos en los conjuntos de datos (anomalías muy concretas y pocos datos normales) nos centraremos en el uso de clasificadores binarios que serán capaces de dar un resultado mucho mejor.

En este trabajo proponemos 2 alternativas para convertir el vector de características en uno más pequeño y más fácil de clasificar. Estas propuestas consisten en usar uno de los 2 métodos de reducción de dimensionalidad más usados, el PCA y los *autoencoders*, para transformar el vector de histogramas en una representación más compacta y con valores más relevantes. De esta forma, no solo podríamos ahorrar espacio de almacenamiento al usar valores más pequeños, si no que podríamos aumentar el número de particiones de los histogramas, reduciendo el sesgo que produce este tipo de representación, y obtener más información sin que el mayor número de variables perjudique a la precisión de nuestro modelo. A continuación pasaremos a discutir ambas posibilidades en mayor detalle.

Análisis de Componentes Principales (PCA)

El análisis de componentes principales es una transformación lineal que traslada los datos a un nuevo sistema de coordenadas. En este nuevo sistema los ejes están ordenados de tal manera que la mayor varianza se produce en la primera coordenada, y la menor en la última. Este orden en los ejes es lo que nos permite usar esta técnica para reducir la dimensionalidad de nuestros datos, usando solamente los primeros ejes que se obtengan. Generalmente es posible mantener más del 95 % de la varianza de los datos con una dimensionalidad mucho menor.

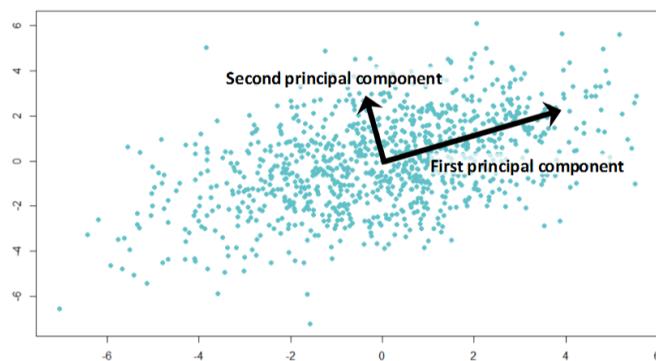


Figura 3.9: Componentes principales de una distribución en 2D.

Un inconveniente de esta técnica es que antes de aplicarla hay que asumir una serie de condiciones que no siempre se dan. Una de estas condiciones es que las variables de entrada sean independientes entre si, lo que no es cierto en nuestro caso, ya que al usar histogramas, habrá relaciones de dependencia entre los valores de un mismo descriptor. Sin embargo, creemos que merece la pena comprobar empíricamente su efecto, dado que en [7] ya se habla de aplicar PCA y se muestra una mejora en el modelo, aunque esta no se exponga como parte de los resultados finales. Además se trata de una técnica de fácil implementación y en ciertos casos se ha conseguido llegar a buenos resultados a pesar de que no se cumpliesen las condiciones teóricas.

Autoencoder

Puesto que la reducción de dimensionalidad podría ayudar a la clasificación y el PCA puede no ser la técnica más adecuada para nuestro problema, proponemos el uso de otra conocida técnica de reducción de dimensionalidad basada en *deep learning*, los *autoencoders*. Ya hemos hablado de ellos en

el apartado de fundamentación teórica, pero recordemos que un *autoencoder* se trata de una red neuronal que se divide en dos partes, un codificador y un decodificador, y que se entrena de manera no supervisada. El codificador reduce la entrada a un tamaño indicado por su capa de código, mientras que el decodificador intenta expandir la información de dicha capa para reconstruir la entrada. Una vez entrenada la red, se extrae el codificador y se usa para reducir la dimensionalidad de los nuevos datos en un vector de características más pequeño pero elaborado.

La gran ventaja de estas redes es su adaptabilidad, ya que puede aplicarse a todo tipo de problemas. Además, puesto que se trata de una red neuronal, podemos ajustar su entrenamiento para que tenga en cuenta la información de clase y favorecer las posibles codificaciones que sean más fáciles de clasificar. Para hacer estoaremos uso del entrenamiento multi-tarea, donde para cada entrada tendremos 2 salidas, cada una con un valor de pérdida propio. La primera salida será la del decodificador, y su pérdida será el valor del error cuadrático al comparar la entrada con la salida obtenida. La segunda salida será una clasificación, al igual que si usaramos una red neuronal normal para clasificar el vector de características. Es importante señalar que usamos esta segunda rama de la red solo para ayudar al codificador, ya que en este caso una SVM es más efectiva a la hora de clasificar. Esta ayuda es importante porque pueden existir muchas formas de codificar una entrada, y solo algunas de ellas serán fáciles de clasificar, si no añadimos información de clase al entrenamiento nos arriesgamos a que la codificación final sea muy precisa y fácil de decodificar pero difícil de usar para nuestro propósito, la clasificación.

El resto del diseño de la red no tiene por qué ser muy complejo ya que no estamos trabajando con datos muy numerosos o complejos. Para el codificador debería bastar con 1 o 2 capas ocultas. Podemos decir lo mismo del decodificador, aunque el clasificador puede ser más pequeño aún ya que trabaja sobre una entrada bastante pequeña que ya está procesada. Un tamaño tan pequeño también nos permitirá no tener que preocuparnos por el rendimiento de la red, que sin duda funcionará a tiempo real.

En cuanto a los inconvenientes de esta propuesta nos encontramos con la mayor complejidad que supone diseñar una red neuronal, especialmente comparandolo con un PCA, y que entrenar este tipo de codificadores suele necesitar más datos que su alternativa, algo de lo que hablaremos en el apartado de resultados.

3.4. Aspectos de implementación

Para implementar este modelo se ha usado Python como lenguaje de programación. Este lenguaje es fácil de manejar y es además ampliamente usado en los campos del aprendizaje automático y de la visión por computador. Una desventaja de esta elección será un tiempo de ejecución mayor que si usásemos un lenguaje de más bajo nivel como C++, sin embargo podemos usar compiladores como numba [14] para mitigar este problema.

Dos de las bibliotecas más importantes que se han usado han sido numpy [15], para el uso de arrays y algunos cálculos numéricos; y opencv [16], para los aspectos relacionados con la visión por computador, como los algoritmos de detección de características, rastreamiento y triangulación de Delaunay.

La implementación de los *autoencoders* se ha llevado a cabo con Keras [17], una biblioteca ampliamente usada en el campo del *deep learning*. El resto de modelos y funciones de aprendizaje automático se han extraído de la librería scikit-learn [18].

Para el diseño del *autoencoder* se ha usado Optuna [19], un *framework* de optimización de hiperparámetros con el que hemos realizado distintas pruebas en las que se probaban diversos valores y arquitecturas para así elegir aquellos que diesen un mejor resultado.

El código implementado para este trabajo se puede encontrar en el siguiente repositorio de GitHub (<https://github.com/DiegoNavaca/TFG.git>). Los archivos están divididos de la siguiente manera:

- **descriptors.py**: incluye todo lo relacionado con la extracción de descriptores de un vídeo. Una vez procesado, sus descriptores serán guardados en un archivo para facilitar su tratamiento y limitar el uso de memoria.
- **visualization.py**: contiene funciones que permiten visualizar distintos aspectos del procesamiento del vídeo, como el grafo de Delaunay o las trayectorias.
- **files.py**: incluye funciones para la lectura del *ground truth* de cada conjunto de datos, así como de las etiquetas de cada vídeo. También incluye una función para extraer los descriptores de un conjunto de vídeos y la preparar las etiquetas de estos de manera acorde.

- **data.py**: incluye funciones que leen la información guardada en los archivos y devuelven el vector de histogramas y etiquetas que se usarán para entrenar el modelo.
- **models.py**: contiene todo lo referente al entrenamiento y evaluación de los modelos de codificación y clasificación, a partir de los histogramas y etiquetas ya preparados.
- **autoencoders.py**: contiene la definición del autoencoder usado para codificar los histogramas, con parámetros para ajustar diversos aspectos de su diseño, como el número de capas o la función de activación de las capas ocultas.
- **main.py**: centro de la implementación. Usa las funciones del resto de archivos con los parámetros adecuados para cada conjunto de datos.
- **optimize_params.py**: contiene la información relacionada con el uso de Optuna para el diseño del *autoencoder*.

3.5. Consideraciones éticas

Aunque este modelo no recurre en ningún momento a la detección o identificación de personas, hay que tener en cuenta que la grabación en vídeo de una persona o grupo de personas se considera como datos de carácter personal que deberán tratarse de manera ética y legal. Dado que estamos trabajando a partir de las relaciones espaciales y temporales de un conjunto de puntos escogidos por su facilidad de rastreamiento, nuestro modelo debería evitar varios de los sesgos asociados a modelos basados en redes convolucionales, como el color de la piel o el género, sin embargo hay que seguir teniendo en cuenta otros sesgos, relacionados no solo con el funcionamiento interno del modelo, si no también con su posible uso, como por ejemplo la diferencia entre las escenas que se usen en su entrenamiento y las que se produzcan en situaciones reales.

Otro factor a tener en cuenta es la función con la que se aplica el modelo, ya que aunque sus principales funciones pueden ser beneficiosas para la sociedad, como detectar de forma temprana una escena de pánico para controlar la situación lo antes posible, también se puede aplicar de maneras contrarias a la ética, especialmente cuando se usa en conjunción con otros sistemas como uno de reconocimiento facial. Un ejemplo de esto podría ser si se usase este modelo (o uno similar) para detectar una manifestación, seguido

de un modelo de reconocimiento facial para identificar a los manifestantes.

En conclusión, creemos un modelo de estas características, capaz de acelerar la respuesta ante situaciones peligrosas, conlleva beneficios que compensan los riesgos, sin embargo es muy importante ser conscientes de estos riesgos y asegurar el nivel ético tanto en el desarrollo como en la implantación de este tipo de modelos.

Resultados experimentales

4

En este apartado explicaremos y comentaremos las pruebas realizadas y sus resultados. También compararemos dichos resultados con los expuestos en [7]. En primer lugar explicaremos el proceso seguido en cada conjunto de datos para obtener los resultados. Después haremos una valoración basada en la calidad de las clasificaciones obtenidas y de la velocidad con la que se han obtenido.

4.1. Metodología usada

La estructura básica que seguiremos en cada conjunto de datos será la misma. En primer lugar extraeremos los descriptores de todos los fotogramas de cada vídeo y los guardaremos en un archivo por cada vídeo. Luego separaremos los archivos en dos conjuntos, uno de entrenamiento y otro de evaluación y trabajaremos solo con el primer conjunto para entrenar nuestro modelo que luego será testeado en el conjunto de evaluación. Para una mayor precisión en los resultados y para imitar el proceso seguido en [7] usaremos validación cruzada, es decir, repetiremos el entrenamiento y la evaluación varias veces, cambiando la división de los archivos en cada ocasión para evitar un sesgo en la partición. Puesto que la extracción de descriptores es independiente para cada vídeo, solo será necesario llevarla a cabo 1 vez.

Para el entrenamiento primero recorreremos los archivos de entrenamiento para establecer el rango de los histogramas, tal y como comentábamos en la descripción del modelo. A continuación generamos los histogramas y los usamos para entrenar una SVM o una de las opciones que hemos expuesto para la reducción de dimensionalidad, PCA o autoencoder. Si nos decantamos por reducir la dimensionalidad, usamos el modelo entrenado para codificar los histogramas y después entrenamos la SVM.

Para la evaluación calcularemos 2 métricas: la precisión y el área bajo la curva (AUC). La precisión es una métrica sencilla que mide el porcentaje de

instancias que han sido clasificadas correctamente. La AUC es una métrica exclusiva de la clasificación binaria en la que se mide el área que queda debajo de la curva que se obtiene cuando representamos el ratio de verdaderos positivos (TPR) en función del ratio de falsos positivos (FPR). La principal diferencia entre ambas métricas se da cuando hay una gran diferencia entre el número de instancias de ambas clases, ya que en ese caso nuestro modelo podría predecir de manera sesgada favoreciendo a la clase con más instancias, concentrando los errores en la que está menos representada. En el caso de que ambas clases estén balanceadas lo normal es que los errores se repartan de forma equitativa entre ambas clases y por lo tanto ambas métricas den resultados similares.

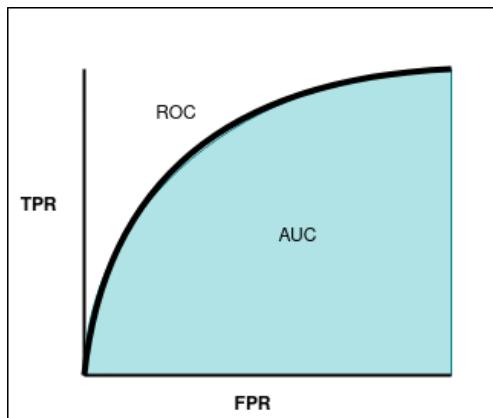


Figura 4.1: Descripción gráfica de la AUC.

En este apartado también hablaremos de la velocidad de nuestro modelo, que mediremos en función del tiempo de extracción de los descriptores, sin tener en cuenta el tiempo que se tardaría en convertir estos descriptores histogramas, codificarlos y clasificarlos. Hacemos esto porque estos últimos pasos tienen muy poco peso computacional y se ejecutan en una cantidad de tiempo despreciable. La única referencia a la velocidad del modelo de [7] es que alcanza los 5 *fps*, pero no se especifican los parámetros que usa para eso, ni el conjunto de datos usado. En nuestro caso haremos un estudio en las 3 escenas del conjunto de datos UMN, variando el único parámetro que afecta significativamente al tiempo de extracción, el umbral usado por el algoritmo FAST. Este umbral determinará el número de puntos con los que trabaja nuestro modelo, y por lo tanto también alargará o acortará nuestro tiempo de ejecución. Intentaremos buscar el valor que maximice los fotogramas por segundo (*fps*) sin provocar una pérdida significativa de la AUC. Dado el ta-

maño del conjunto de Violent Flows, no haremos este experimento en él, pero sí indicaremos los *fps* que se han alcanzado durante el experimento.

4.1.1. UMN Crowd Dataset

Como decíamos antes, este conjunto de datos se centra en la detección de escenas de pánico. Cada vídeo empieza con un grupo de personas caminando de manera normal hasta que reciben una señal, momento en el que todos echan a correr y empieza la anomalía. Esta anomalía termina cuando no se vea a nadie corriendo, bien porque todo el mundo ha salido del plano de la cámara (lo más común), o bien porque la gente deja de correr.

Este conjunto contiene 3 escenas distintas que trataremos de forma independiente y 11 vídeos en total. La primera escena contiene solo 2 vídeos, la segunda 6 y la tercera 3. Puesto que el número de vídeos en cada escena es bajo usaremos un vídeo para evaluar el modelo y el resto para entrenarlo, repitiendo el proceso tantas veces como vídeos haya. Este tipo de validación cruzada se conoce como *leave – one – out*. La clasificación que haremos será a nivel de fotograma, es decir, clasificaremos cada fotograma del vídeo, a excepción de los primeros L fotogramas de los que no tenemos información. En este caso el número de fotogramas normales es mucho mayor que el de anómalos, por lo que la métrica AUC nos será de gran ayuda.

Si comparamos las 3 escenas podremos apreciar algunas diferencias. La más distintiva sin duda es la escena 2 que se sitúa en un pasillo, con movimientos constantes de luces y sombras a través de las ventanas y puertas, que sin duda afectarán al modelo. Además, el ángulo en el que está colocada la cámara en esta escena provoca que la mayoría de los desplazamientos sean acercándose o alejándose de esta, lo que dificulta medir las distancias recorridas. Otro diferencia es que en esta escena hay un gran número de personas estáticas en un solo lugar, ya que en las otras dos escenas todo el mundo se está desplazando de un lugar a otro. Por último es importante señalar que esta escena es la única en la que se puede considerar que la anomalía termina antes de que todo el mundo haya abandonado el plano de la cámara, ya que en algunos vídeos parte de la multitud se dirige al fondo del pasillo y deja de correr antes de que se acabe el vídeo. Esto último será un problema a tener en cuenta ya que la frontera entre la anomalía y la vuelta a la normalidad es difusa y será una gran fuente de ruido para nuestro modelo.

Las escenas 1 y 3 son más similares entre sí. Ambas se sitúan al aire li-

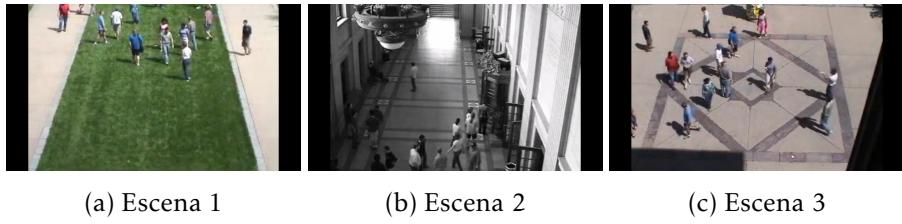


Figura 4.2: Fotogramas de ejemplo de las 3 escenas del conjunto UMN.

bre, por lo que las únicas sombras en movimiento son las proyectadas por las propias personas que forman la multitud, por lo que no deberían afectar negativamente al modelo. Además al ser un espacio abierto la multitud es más propensa a moverse de forma paralela a la cámara, haciendo que sea más fácil estimar velocidades y distancias.

Como decíamos en el apartado de descripción del problema, las marcas de inicio y final de la anomalía del vídeo no son precisas, por lo que usaremos el inicio indicado en [7] y tendremos que establecer el final manualmente. Para marcar el final de estas anomalías se ha buscado un frame en el que, o bien la multitud haya salido de la escena, o bien haya dejado de correr. El *ground truth* que se ha usado para generar las etiquetas es el que se muestra en la tabla 4.3. Como se puede ver, en algunos casos el fin de la anomalía coincide o es muy cercano al final del vídeo, pero en otros hay una diferencia significativa, por lo que es extraño que en [7] solo se indiquen los valores de inicio.

Vídeo	Inicio anomalía	Fin anomalía	nº fotogramas
UMN1	484	610	622
UMN2	665	806	825
UMN3	303	498	546
UMN4	563	672	681
UMN5	492	701	765
UMN6	450	553	576
UMN7	734	891	891
UMN8	454	616	666
UMN9	551	654	654
UMN10	570	663	672
UMN11	717	797	804

Figura 4.3: *Ground Truth* del conjunto de datos UMN

4.1.2. Violent Flows

Como decíamos en la descripción del problema, este conjunto de datos se centra en la detección de escenas violentas. Tenemos una selección de vídeos de multitudes, generalmente celebrando o animando a un equipo deportivo, y en donde no se aprecia violencia; y un grupo de vídeos en los que se ven peleas de distinto tipo, y que serán marcados como anómalos.

En este caso la clasificación se hace a nivel de vídeo, es decir, combinamos la información de todo el vídeo para clasificarlo como una situación normal o anómala. Para hacer esto, dado que se trata de vídeos muy cortos (3.6 seg de media), calcularemos la media de todos los histogramas. El conjunto consta de 246 vídeos donde 125 son normales y 121 son anómalos. Puesto que ambas clases están equilibradas usaremos una validación cruzada de 5 pliegues, igual que en [7], mezclando todos los vídeos de manera aleatoria.

4.2. Comparación de resultados

En este apartado expondremos y comentaremos los resultados obtenidos. Compararemos 3 modelos distintos: uno que no codifica el vector de características, uno que usa un PCA y uno que usa un *autoencoder*. Usaremos los resultados de [7], a los que llamaremos LMVD (Local Mid-Level Visual Descriptors), como referencia ya que, como decíamos en el apartado 3, es en ese modelo en el que nos hemos basado.

Fijando una semilla para la generación de numeros aleatorios es posible replicar fácilmente los resultados del modelo sin codificar y del que usa PCA, sin embargo, el entrenamiento de una red neuronal provoca que los resultados del modelo con *autoencoder* varíen en mayor medida entre ejecuciones, por lo que en su caso calcularemos la media de 5 ejecuciones. Hacemos esto en vez de aumentar el número de pliegues porque eso le daría un conjunto de entrenamiento más grande, invalidando la comparación.

Primero trataremos los resultados cualitativos medidos por la precisión y la AUC, y después hablaremos de la velocidad de nuestro extracto de descriptores comparado con la del modelo original. Estas pruebas se han realizado en un portatil con un procesador i3 de 2.2GHz y la implementación se ha hecho en python con el apoyo de la función *jit* del módulo *numba* [14] para acelerar los cálculos de las funciones matemáticas. Dada la baja potencia del

procesador usado es de esperar que estos resultados puedan mejorar en una situación real con un *hardware* adecuado.

4.2.1. UMN Crowd Dataset

Como comentábamos antes, trataremos cada escena de este conjunto de datos de manera independiente, por lo que tenemos que estudiar 3 casos diferentes. Es importante recordar que el número de fotogramas no es especialmente grande (especialmente en las escenas 1 y 3), por lo que pequeñas diferencias en los resultados pueden no ser significativas. Además puesto que el inicio y el final de las anomalías no se puede determinar de manera exacta, es de esperar cierta incertidumbre en los fotogramas que transicionan de un estado a otro.

Modelo	Precisión	AUC
LMVD	n/a	0.996
Sin codificación	0.990	0.987
PCA	0.988	0.974
Autoencoder	0.979	0.967

Figura 4.4: Resultados en la escena 1 de UMN

En los resultados de la primera escena que podemos ver en 4.4 se puede apreciar que el modelo propuesto obtiene unos resultados muy cercanos a los de [7]. Dado que estamos hablando de solo 2 vídeos, la diferencia entre nuestro modelo sin codificación y el LMVD no parece significativa, ya que podría deberse al azar o a ligeras diferencias en la manera etiquetar los fotogramas. En los modelos que codifican los datos sí que empezamos a ver una diferencia, muy probablemente debido a que tenemos muy pocos datos para entrenar adecuadamente nuestros codificadores y a que la precisión de base ya es especialmente alta.

Modelo	Precisión	AUC
LMVD	n/a	0.986
Sin codificación	0.961	0.943
PCA	0.952	0.944
Autoencoder	0.957	0.946

Figura 4.5: Resultados en la escena 2 de UMN

En la segunda escena (4.5) podemos ver un resultado muy distinto al de las otras escenas, con una pérdida bastante notable con respecto a la indicada en [7]. Podemos atribuir esta pérdida a muchos factores, como la iluminación que produce luces y sombras que afectan al detector de puntos y al rastreador o el poco movimiento de los participantes, que dificulta ajustar el umbral para el filtro de puntos que hacemos en función de su velocidad. Sin embargo, uno de los mayores problemas que nos encontramos en esta escena es que es la única en la que las personas dejan de correr antes de salir del plano de la cámara, obligandonos a determinar de manera arbitraria el momento en el que cesa la anomalía. Dado que desconocemos los índices que se han usado en [7] para marcar dicho final de la anomalía es difícil replicar fielmente el experimento realizado y comprobar dónde falla, si ese fuera el caso, nuestro modelo. En cuanto a la comparación entre nuestros 3 modelos podemos ver que aunque las diferencias son muy pequeñas como para poder afirmar con confianza que hay una mejora significativa, sí que podemos ver que cuando se proporcionan más datos a los codificadores, el resultado mejora con respecto a lo visto en la primera escena.

Modelo	Precisión	AUC
LMVD	n/a	0.975
Sin codificación	0.987	0.973
PCA	0.986	0.972
Autoencoder	0.984	0.984

Figura 4.6: Resultados en la escena 3 de UMN

En la última escena (4.6) empezamos a encontrar resultados más alentadores. Por un lado, la diferencia entre los resultados obtenidos en el modelo sin codificar y el que usa PCA con los expuestos en [7] es claramente despreciable. Por otro lado, los resultados al usar un *autoencoder* parecen ser mejores que los demás, tal y como esperábamos, dado que esta escena tiene más datos que la 1, pero sin los problemas de la 2.

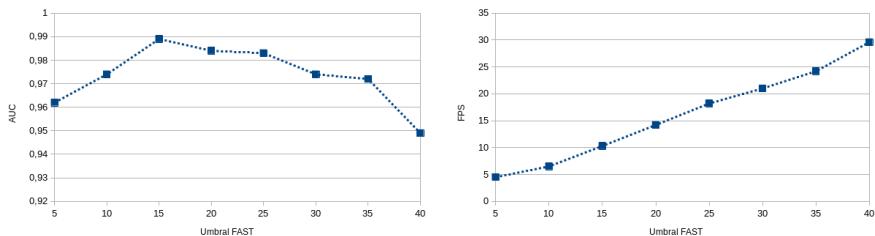
Si hacemos una media de los tres resultados (4.7) podemos ver que el modelo de [7] consigue un mejor resultado que nuestra implementación, sin embargo es importante destacar que esta diferencia se debe principalmente a los resultados de la escena 2, que como ya se ha comentado antes, distan bastante de lo que cabría esperar. Por otro lado, cuando comparamos nuestros 3 modelos puede parecer que las diferencias entre unos y otros son irrelevan-

Modelo	Precisión	AUC
LMVD	n/a	0.985
Sin codificación	0.980	0.968
PCA	0.975	0.963
Autoencoder	0.973	0.966

Figura 4.7: Media de los resultados en las 3 escenas.

tes, sin embargo, de manera similar al *No Free Lunch Theorem* que comentábamos en el apartado de teoría, cuando observamos los resultados en una escena concreta sí que se aprecian las diferencias. De esta forma, aunque la media al usar autoencoder similar a la de las otras propuestas, en la escena 3 veíamos que al tener más datos, el autoencoder superaba, no solo estas, si no al modelo de [7]. De igual manera, en la escena 1 en la que teníamos pocos datos la propuesta que mejor resultado daba era aquella que no codificaba los datos.

En cuanto a la velocidad de extracción de los descriptores, como podemos ver en 4.8 y 4.10 la AUC oscila dentro de los valores esperados cuando se usa un umbral de entre 10 y 35, lo que nos deja con una velocidad máxima de entre 20 y 25 fotogramas por segundo sin una pérdida notable de la fiabilidad del modelo. Este valor alcanzado es muy superior a los 5 *fps* de los que se habla en [7], casi alcanzando el punto de ejecución en tiempo real, aún siendo una ejecución de una implementación en Python y en un ordenador de gama media.

Figura 4.8: Relación del umbral de FAST con la AUC (izquierda) y con los *fps* (derecha) en la escena 1 de UMN

Por otro lado, en la escena 2 (4.9) volvemos a ver un comportamiento extraño en cuanto a la velocidad de extracción y al umbral óptimo para FAST. Como podemos ver, los mejores resultados se obtienen con un umbral muy

bajo, sin embargo la velocidad de extracción no baja de los 5 fps . Además, si estamos dispuestos a renunciar a parte de la precisión del modelo podemos alcanzar velocidades muy altas, de más de 30 fps . Todo esto se puede deber a 2 factores que ya hemos comentado anteriormente. Por un lado, el poco movimiento de las personas en escena puede obligarnos a usar un umbral menor que nos permita captar mejor detalles que de otra forma pasarían desapercibidos. Por otro lado, los relativamente largos períodos en los que solo se ve a una o dos personas en pantalla, cuando el resto han abandonado el plano pueden llevar a una menor cantidad de puntos a rastrear durante un gran número de frames, bajando notablemente el tiempo de extracción medio de los descriptores y por lo tanto dando la impresión de una mayor velocidad general. Independientemente de las razones, la velocidad se sigue manteniendo por encima de los 5 fps por lo que el modelo sigue comportándose de manera adecuada.

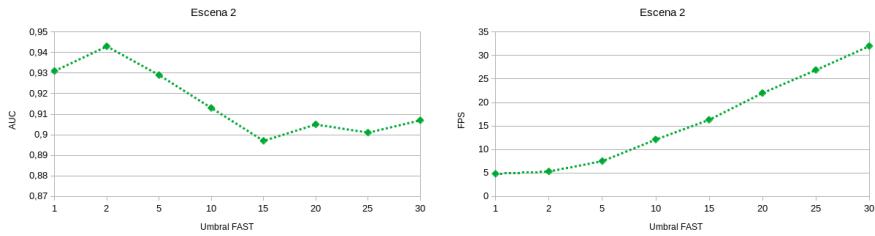


Figura 4.9: Relación del umbral de FAST con la AUC (izquierda) y con los fps (derecha) en la escena 2 de UMN

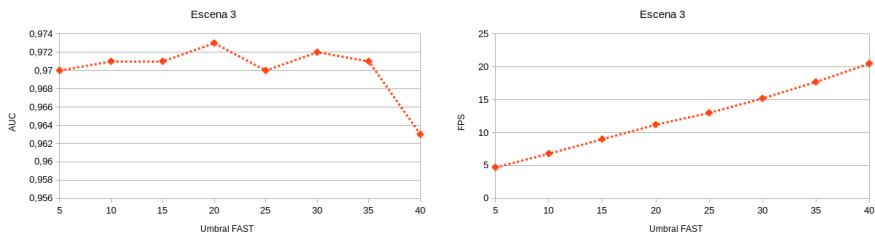


Figura 4.10: Relación del umbral de FAST con la AUC (izquierda) y con los fps (derecha) en la escena 3 de UMN

4.2.2. Violent Flows

Los resultados obtenidos en este conjunto de datos son los que se pueden ver en 4.11. Estos resultados se han obtenido con un umbral FAST de 10, lo

que nos daba una velocidad de extracción de los descriptores de *7 fps*. Esta velocidad es inferior a la que cabría esperar de una escena normal, como las del conjunto UMN, dado que la cámara se mueve en muchos vídeos, inutilizando el filtro por velocidad que usamos para reducir el número de puntos a seguir. Algunos vídeos también presentan multitudes muy densas en las que se detectan una enorme cantidad de puntos a seguir, dificultando más aún el procesamiento de los datos. Puesto que aún en unas condiciones tan poco favorables se consigue una mejora en la velocidad con respecto a [7], se valora muy positivamente el rendimiento del modelo.

Modelo	Precisión	AUC
LMVD	0.844	0.880
Sin codificación	0.862	0.867
PCA	0.894	0.898
Autoencoder	0.870	0.870
Clasificador Autoencoder	0.804	0.803

Figura 4.11: Resultados en la escena 3 de UMN

En cuanto a la calidad de los resultados podemos ver que el modelo sin codificar y el que usa un *autoencoder* se quedan ligeramente por debajo de la AUC del modelo original, sin embargo lo superan notablemente en la precisión. Este conjunto de datos no presenta una gran diferencia entre el número de instancias en una clase u otra, por lo que llama la atención que haya tanta diferencia entre la precisión y la AUC que [7] afirma conseguir, sin embargo no tenemos información suficiente como para valorar esto. En cuanto al modelo que usa PCA podemos ver que mejora muy notablemente a los otros dos, e incluso llega a superar la AUC del modelo LMVD.

En cuanto a la razón de por qué vemos una mejora tan sustancial con PCA pero no con el *autoencoder*, podría deberse a que este conjunto de datos contiene una cantidad muy pequeña de instancias (solo 246), por lo que podemos entrenar correctamente un PCA pero no tenemos suficiente como para entrenar un buen *autoencoder*. Esto además se ve reforzado por la gran variación entre las ejecuciones del modelo con *autoencoder*, cuyos resultados pueden variar entre una AUC de 0.85 hasta una de 0.89. Otra razón para señalar la falta de datos es que el clasificador que usamos para ayudar en el entrenamiento del autoencoder es capaz de obtener valores de AUC casi perfectos en el conjunto de entrenamiento, sin embargo cuando se aplica sobre

nuevos datos pierde mucha precisión, como se puede ver en 4.11. Este ultimo resultado también nos sirve para justificar el uso de una SVM encima del codificador, ya que una red neuronal funciona peor que el modelo propuesto, probablemente por las mismas razones por las que el *autoencoder* no llega a rendir como cabría esperar, es decir, por la falta de datos.

Conclusiones y trabajo futuro



5.1. Conclusiones

En este trabajo hemos visto cómo podemos usar modelos de aprendizaje automático para detectar peleas o escenas de pánico. Además hemos comprobado que añadir un modelo que codifique los datos antes de clasificarlos puede facilitar la separación en clases de estos, permitiéndonos alcanzar un mejor resultado.

Uno de los mayores problemas con el que nos hemos encontrado ha sido la falta de datos, especialmente para el entrenamiento del codificador, que en más de un caso ha limitado nuestras opciones para el procesamiento de dichos datos, debido al *overfitting*.

Para evaluar la calidad del modelo propuesto hemos comparado sus resultados con el modelo en el que nos habíamos basado ([7]). Los resultados obtenidos en el primer conjunto de datos (UMN) no han conseguido superar la calidad de la clasificación (medida según el AUC) del modelo de referencia, principalmente debido a unos resultados por debajo de lo esperado en la segunda escena, sin embargo sí que han conseguido superar con creces la velocidad de ejecución de este, llegando a alcanzar los 20-25 *fps* sin una perdida notable en la métrica de referencia.

En el segundo conjunto de datos sí que hemos conseguido una mejora notable con respecto al modelo de referencia. En este conjunto, al tener en cuenta las 2 métricas usadas, 2 de los modelos propuestos alcanzaban una mejor precisión, a pesar de una peor AUC. Dado que dicho conjunto de datos no presentaba una gran diferencia en el número de instancias de una clase con respecto a la otra, no podemos ignorar la métrica de la precisión, y por lo tanto valoramos positivamente los resultados obtenidos en estos dos modelos. En el tercer modelo, los resultados son claramente superiores a los de el modelo de referencia, superandolo en ambas métricas, lo que nos muestra

la capacidad de los codificadores (en este caso PCA) para facilitar la clasificación de los datos.

5.2. Trabajo futuro

En cuanto al trabajo que se podría realizar en el futuro, creemos que el mayor problema actualmente para el desarrollo de modelos que detecten anomalías en multitudes es la falta de buenos conjuntos de datos, que sean fácilmente accesibles y completos.

Como ya hemos mencionado en el apartado anterior, el bajo número de instancias con las que hemos tenido que trabajar ha llevado a un notable *overfitting* tanto a la hora de codificar los datos, como a la hora de clasificarlos.

Por otro lado, los *datasets* usados se centraban en la detección de un tipo de anomalía concreta, obligandonos a tratar este problema como uno de clasificación. Sería muy útil poder contar con un conjunto de datos más general que nos permitiese investigar modelos de una sola clase, que podrían ser más aptos para su implementación en la sociedad. En un caso real no se tendrán muchos ejemplos de anomalías con los que entrenar un modelo, por lo que poder llevar a cabo este entrenamiento solo con vídeos de escenas normales es altamente deseado.

Bibliografía

- [1] Tom M. Mitchell. *Machine Learning*. New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.
- [2] David H. Wolpert y et al. *No Free Lunch Theorems for Optimization*. 1997.
- [3] Edward Rosten, Reid Porter y Tom Drummond. “Faster and Better: A Machine Learning Approach to Corner Detection”. En: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.1 (2010), págs. 105-119. doi: [10.1109/TPAMI.2008.275](https://doi.org/10.1109/TPAMI.2008.275).
- [4] UMN Crowd Dataset. http://mha.cs.umn.edu/proj_events.shtml#crowd/.
- [5] Violent Flows Dataset. <https://www.openu.ac.il/home/hassner/data/violentflows/>.
- [6] Duan-Yu Chen y Po-Chung Huang. “Motion-based unusual event detection in human crowds”. En: *J. Visual Communication and Image Representation* 22 (feb. de 2011), págs. 178-186. doi: [10.1016/j.jvcir.2010.12.004](https://doi.org/10.1016/j.jvcir.2010.12.004).
- [7] Hajar Fradi, Bertrand Luvison y Quoc-Cuong Pham. “Crowd Behavior Analysis Using Local Mid-Level Visual Descriptors”. En: *IEEE Transactions on Circuits and Systems for Video Technology* PP (oct. de 2016), págs. 1-1. doi: [10.1109/TCSVT.2016.2615443](https://doi.org/10.1109/TCSVT.2016.2615443).
- [8] R. Mehran, A. Oyama y M. Shah. “Abnormal crowd behavior detection using social force model”. En: (2009), págs. 935-942.
- [9] Mahdyar Ravanbakhsh y col. “Plug-and-Play CNN for Crowd Motion Analysis: An Application in Abnormal Event Detection”. En: (oct. de 2016).
- [10] Mahdyar Ravanbakhsh y col. “Training Adversarial Discriminators for Cross-channel Abnormal Event Detection in Crowds”. En: (jun. de 2017).

- [11] Tobias Senst, Volker Eiselein y Thomas Sikora. “Robust Local Optical Flow for Feature Tracking”. En: *IEEE Transactions on Circuits and Systems for Video Technology* 22 (sep. de 2012). doi: [10.1109/TCSVT.2012.2202070](https://doi.org/10.1109/TCSVT.2012.2202070).
- [12] Carlo Tomasi y Takeo Kanade. *Detection and Tracking of Point Features*. Inf. téc. International Journal of Computer Vision, 1991.
- [13] Michael Hahsler, Matthew Piekenbrock y Derek Doran. “dbscan: Fast Density-Based Clustering with R”. En: *Journal of Statistical Software* 91.1 (2019), págs. 1-30. doi: [10.18637/jss.v091.i01](https://doi.org/10.18637/jss.v091.i01).
- [14] LLVM '15: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052.
- [15] Charles R. Harris y col. “Array programming with NumPy”. En: *Nature* 585.7825 (sep. de 2020), págs. 357-362. doi: [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2). URL: <https://doi.org/10.1038/s41586-020-2649-2>.
- [16] G. Bradski. “The OpenCV Library”. En: *Dr. Dobb's Journal of Software Tools* (2000).
- [17] Francois Chollet y col. *Keras*. 2015. URL: <https://github.com/fchollet/keras>.
- [18] Fabian Pedregosa y col. “Scikit-learn: Machine learning in Python”. En: *Journal of machine learning research* 12.Oct (2011), págs. 2825-2830.
- [19] Takuya Akiba y col. “Optuna: A Next-generation Hyperparameter Optimization Framework”. En: *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.